Agile Codeline Management

Steve Berczuk
Independent Consultant

Software Developers often view version management tools and techniques as a necessary evil. This is particularly true of developers practicing agile techniques. However, Version Management properly applied can be an aid to agility rather than something that gets in the way.

A version of this article appeared as a StickyMinds Original article on May 31, 2003. http://www.stickyminds.com/r.asp?F=DART_6484.

Software configuration management (SCM) techniques have a reputation as something to tolerate as a necessary evil, in spite of the fact that they appear to make development more complicated. With time developers see the benefit of version control, but often they have the impression that SCM is "important" in some not understood way. The fear is particularly acute when it comes to codeline management, especially when the word "branch" is mentioned.

In this article I will explain the concept of Agile Software Configuration Management, and give an example of how one of the more feared codeline management concepts the branch, can be understood as a tool for agility. This is one example of why version management is part of the software developer's toolkit, even the agile developer's toolkit.

What is Agile Software Development?

In his book, *Agile Software Development Ecosystems*, Jim Highsmith says that "Agility is the ability to both create and respond to change in order to profit in a turbulent business environment." A team that practices Agile Software Development is one that is responsive rather than weighed down by process and which embraces these values (From the Agile Manifesto: www.agilemanifesto.org):

- To Value individuals and interactions processes and tools
- To Value working software over comprehensive documentation
- To Value customer collaboration over contract negotiation
- To Value responding to change over following a plan

Extreme Programming and SCRUM are two of the popular methods for agile software development.

Agile and SCM

Developers often have a fear that SCM techniques beyond basic version management adds complexity to the development process and slows down progress. It is true that inappropriate use of SCM techniques, such as branching, can do more harm than good to

your project. The correct application of SCM tools and techniques can give you the courage to forge ahead quickly since you know that there will be little risk that you will get lost.

SCM is often associated with heavyweight "Process." When you apply SCM appropriately it becomes a tool to enable your approach to software development to work more effectively.

Agile SCM applies the values of the Agile Manifesto in this way:

- Individuals and Interactions over Processes and Tools
    - SCM Tools should support the way that you work, not the other way around. Often the reason behind frustrating SCM processes is trying to fit a tool that imposes a process into an organization that needs a much different kind of process.
- Working Software over Comprehensive Documentation
    - Executable Knowledge over Documented Knowledge. SCM can automate development policies and processes, so you can both make things easy for developers and ensure that necessary procedures are followed by writing scripts instead of documenting procedures.
- Customer Collaboration over Contract Negotiation
    - SCM should facilitate communication among stakeholders and help manage expectations. By using the appropriate codeline structures you can keep make it easier to create snapshots of the current state of the system and make it easier for stakeholders to see them.
- Responding to Change over Following a Plan
    - SCM is about facilitating change, not preventing it. Create codeline structures that isolate the components that need to be kept stable from those that are in active development; don't make the entire team move at the pace of the more restrictive components.

To understand how to structure codelines for agility you need to understand the concepts codeline and codeline policies.

Agile Codelines

A *codeline* is a progression of the set of source files and other artifacts that make up some software component as it changes over time. You can have multiple codelines for a project that are independent of each other. For example, you can do your development work on one codeline (The Mainline), and managed library code from other vendors on a separate codeline (Third Party Codeline). When you create your workspace for development you would have a script that checks-out the appropriate versions of the appropriate components from each codeline into your workspace.

Each codeline has a *codeline policy* associated with it. The codeline policy describes how the codeline is to be used. This includes things like:

- How often to check in changes.
- How much testing and validation is required before a check-in.
- Who can check in changes and when.

For example, during development on the Mainline, you may encourage people to check things in frequently, but you want to control changes to the Third Party Codeline more closely.

There are many approaches to managing codelines and deciding how few, or how many codelines to have for a project. The most effective, approach for agile environments is to do all of your work on a single codeline, a Mainline, perhaps supplemented by some components from a Third Party Codeline.

Mainline development by itself is not enough for agile development though. Agile development requires frequent integration. If you have a policy of maintaining a Mainline of shippable quality at all times, you may find that the cost of checking-in code is high enough that developers will check-in only once or twice a day. For example, if you verify quality by an exhaustive set of tests that take an hour to run, there is an incentive to code as much as possible before starting the test, say before going out to lunch. (This also means the there were a lot of changes in each checkin, making it harder to determine what broke.) If you take the further step of making sure that only one person is checking code in a a time using a semaphore mechanism, you could be locking out the entire team.

An Active Development Line approach has advantages if you don't need to be 100% sure that the codeline is perfect on a minute to minute basis. With the appropriate use of simple tests like Unit Tests and Smoke Tests, developers can check code in quickly, and exhaustive tests such as Regression Tests can run on an integration machine periodically.

Branches

Single Active Development Line has many advantages in terms of simplicity, consistency, and minimizing duplication. There would be many advantages to always delivering a product off of the current active codeline. There are some situations where you discover that there are benefits to having a version of the code evolve independently for a strictly limited time. This is where creating a branch is useful, especially in an agile environment (or an environment trying to be more agile).

A common branching pattern is a Release Line. Consider a scenario where you deliver an application to the customer, and then start work on the next version. In spite of what you imagined to be thorough testing, your customer finds a critical problem in the code. You have three choices: ignore the problem until the next release is done, get the current codeline in shape to ship the customer a new version based, or provide the customer with a fix based on the version that they have. Each solution has merits and disadvantages, but you should make your decision based on the needs of the situation rather than fear.

Ignore It

If you have a frequent enough release cycle it may be feasible to ask the customer to wait until the next release. Whether this is feasible depends on the severity of the issues,

whether there is any procedural work around that the customer can do, and how much good will you may lose with the customer.

Provide a fix based on the mainline

If waiting until the next release is not a viable option you need to provide a fix. One way is to ship a version of the mainline after verifying that the mainline code is compatible with what the customer has now, and also after ensuring that the same problem does not occur in the mainline. There are many advocates of this approach, especially among those who advocate agile software development. Deciding whether this is a good solution to the problem depends on a number of social and engineering issues that involve what the customer is willing to accept and how much the architecture supports it.

Branch

The third option is to provide a fix based on the code that you shipped to the customer. When you ship a product you can create a Release Line that represents the evolution of that version of the code. A Release Line is active only as long as you need to support that release, typically until the next release ships. If all goes well you may make no changes to a release line. In addition to being based on the shipped codebase, a release line typically has a stricter codeline policy that requires more exhaustive testing and verification before changes are checked into it.

A release line frees you by letting you continue to move forward with the current codeline while still being able to address customer needs.

The Trouble with Branches

Working off a branch is not without its costs. One concern, especially in the agile community, is that a release line is “more work” since you may have to make changes in two places: the Mainline and the Release Line. In addition to this people often associate a branch with a messy merge process. In some cases you may not need to merge, or even duplicate changes. If you are following agile development practices, you may find that you have done a good deal of refactoring so that a merge does not make sense because the code bases have diverged so much. Indeed you may have made enough changes to the mainline the customer issue with the release may be irrelevant to the mainline.

To keep a Release Line manageable keep it short lived. Work off of the Release Line only when necessary. If there are other ways to support your application use them. Consider branching only when it would be more disruptive to provide support using the Mainline. Limit how long you provide support for a give customer release. Maintainence and support for a number of old releases can be a drain on resources. The best thing to do is to encourage your customers to upgrade. Luke Hohmann discusses this in *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. Solutions to this approach can involve architecture, marketing, or quality.

Conclusions

In some cases, creating a branch is the "simplest thing that could possibly work," and is appropriate, even when you are doing Agile Software Development or Extreme Programming. The key is to understand why you are branching, and what other solutions could work instead, and what the relative costs are.

Acknowledgements

Brad Appleton and Ron Jeffries gave me valuable feedback on this article.

Biography:

Biography: Steve Berczuk has been developing object-oriented software applications since 1989. Steve is the author (with Brad Appleton) of the book *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, published by Addison-Wesley. He has an M.S. in Operations Research from Stanford University and an S.B. in Electrical Engineering from MIT. You can contact Steve at steve@berczuk.com. His web site is www.berczuk.com.